Università di Pisa

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE
Corso di Laurea Specialistica in Ingegneria Informatica

MASTER THESIS

Porting of the Netmap framework to Windows operating system

Supervisors
Prof. Luigi Rizzo
Prof. Giuseppe Lettieri

Candidate
Alessio Faina

Academic year 2014-2015

Index

Abstract.....1

1. The Netmap framework.....3
1.1 The Netmap architecture.....3
1.2 The Netmap API.....4
1.3 VALE switch and Netmap pipes.....5

2. The Netmap core kernel module.....6
2.1 Necessary steps to make user applications and kernel modules communicate.....6
2.2 IOCTL codes specifications.....7
2.3 Passing data back and forth between kernel and user-space.....8
2.3.1 Memory descriptor list (MDL).....10
2.4 Port of the poll mechanism.....11

3. Generic NIC hook, the NDIS 6 LWF kernel module.....14
3.1 Windows network stack.....14
3.2 Network packet data representation in NDIS 6.....18
3.3 NDIS6 LWF module.....20
3.4 The FUNCTION_POINTER_XCHANGE structure.....22
3.5 Attaching to an interface.....23
3.6 Packets capture.....23
3.7 Packets injection.....25
3.8 Driver unload and clean-up.....29

4. Porting of kernel structures and routines.....32
4.1 Dynamic memory allocation.....32
4.2 Mutexes, spin-locks and read/write locks.....33

5. User-space applications.....36
5.1 Packet-gen.....36
5.2 netmap_user.h header modifications.....36
5.3 Netmap loader utility.....37
5.3.1 Hot install procedure 38
5.3.2 Hot uninstall procedure 38

6. Developing environment and build output.....40
6.1 Output of the build process.....40
6.2 INF Files.....41
6.2.1 netmap.inf.....41
6.2.2 nm-ndis.inf.....41
6.3 Digital module signature.....42

7. User instructions.....44
7.1 Setting up the system.....44
7.2 Installing the modules.....45
7.2.1 Netmap 'core' module install procedure.....45
7.2.2 Netmap NDIS module install procedure.....46
7.3 Using packet-gen.....46

Conclusions.....47
8.1 Performances.....47
8.1.1 Performances of the Netmap core.....48
8.1.2 Tests of TX/RX of data over a 1GbE link.....49
8.1.3 Comparison with different user-space applications.....51
8.1.4 Tests of tx/rx of data over a 10GbE link.....52
8.1.5 Overall comparison between different OS.....55
8.2 Future works.....57

Bibliography.....58

Abstract

Netmap is a framework for high speed packet I/O that can be used by user-space programs needing a boost of throughput performances compared to the ones achievable using the standard network stack provided by a general purpose operating system.

A standard network stack presents high performance limitations when the packet rate starts to increase, for example while trying to use a 10Gbit/s interface at its full speed; the performances are capped by the overhead introduced by the various layers present in the stack, per-packet processing operations and system calls. Using Netmap an application doesn't encounters these issues, because the framework it's designed to bypass the OS network stack as much as possible, and to limit the number of system calls from user space. Other than these two design specifications, various boosting techniques are used to decrease per-packet processing cost.

This approach dramatically increases the overall performances of the packet I/O speed, making possible to achieve line rate even with small packets.

The objective of this thesis has been to port the Netmap framework from FreeBSD and Linux to Windows operating system.

Given that the core of the code is written to be platform independent, one of the main focuses of this work has been to develop the port in a non-intrusive way: for the code to be compliant with the Windows kernel specifics, some headers and platform-specific sources has been created to remap the original FreeBSD functions and types already present.

Under Windows OS, the design choice led to divide the Netmap framework in two distinct modules: a ‘core’ module, and an *NDIS 6 Lightweight Filter* module. The core module is the one containing all the logic that stands in the very basis of the framework.

Moreover, is the point of access for user-space applications that want to use the

Netmap framework. The NDIS module covers the features needed to use any real world network interface card present in a system with Netmap: this is made possible because this sub-family of kernel modules is designed to be attached between the *miniport* level and the *protocol* level of the Windows NDIS network stack. By operating at this level of the network stack, this type of kernel module has the ability to intercept network frames arriving from the OS and from installed NICs, and to inject user-generated frames into installed NICs or into the OS data path.

In addition to the Netmap kernel modules, the packet-gen user-space application has also been ported to Windows: pkt-gen is an software provided with the Netmap suite used to test the speed of a network link by injecting packets as fast as possible.

The port of the kernel modules has been made using Microsoft Visual Studio in conjunction with the Driver Development Kit (DDK), freely available and distributed by Microsoft.

The port of the user-space application has been done with the support of Cygwin, a DLL that provides a set of API to emulate a POSIX compliant environment inside a Windows machine: moreover Cygwin provides a set of headers used to build POSIX compliant applications and a shell-like terminal window where POSIX programs like the GCC compiler and MAKE can be run.

Even though POSIX compliant kernels and the Windows kernel have little to none in common, the port has been successfully completed: furthermore the performances shown during the tests on Windows, confirm that Netmap helps in enhancing the speed of packet I/O respect of using the standard network stack.

Chapter 1

The Netmap framework

General purpose OS are the majority of the used operating systems in production environment: this kind of operating systems are really flexible in every aspect and are really cheap considering the trade-off between performances and costs.

Unfortunately, having the positive aspect of being flexible and so being adaptable to almost every situation makes a general purpose OS to lack of optimized performances: the aspect that the Netmap framework [1] covers is to optimize the packet processing operations in order to improve the performances in these operative systems.

1.1 The Netmap architecture

This optimization is accomplished by Netmap by detaching a network interface from the kernel network stack and linking it directly with user-space applications but leaving the OS in its original state: this kind of architecture leaves to the operating system the managing functions to interact with the hardware like turning the NIC on and off or setting its attributes.

Other than this Netmap exposes to user-space applications three key structures:

- Netmap interface descriptor (*netmap_if* in Fig. 1.1): the representation of an interface that contains information about the name of the interface, its properties

and the mechanisms to reach the Netmap rings pointers from user-space.

- Netmap ring (*netmap_ring* in Fig. 1.1): a replica of a real ring provided by a NIC: it contains an array of slots, the number of free slots and the necessary indexes to manage the transmission and the reception of packets. The array of slots is designed as a circular buffer.
- Netmap slot (*netmap_slot* in Fig. 1.1): is a device-independent buffer that contains data of a packet, its length and a flag to describe the status of the slot.

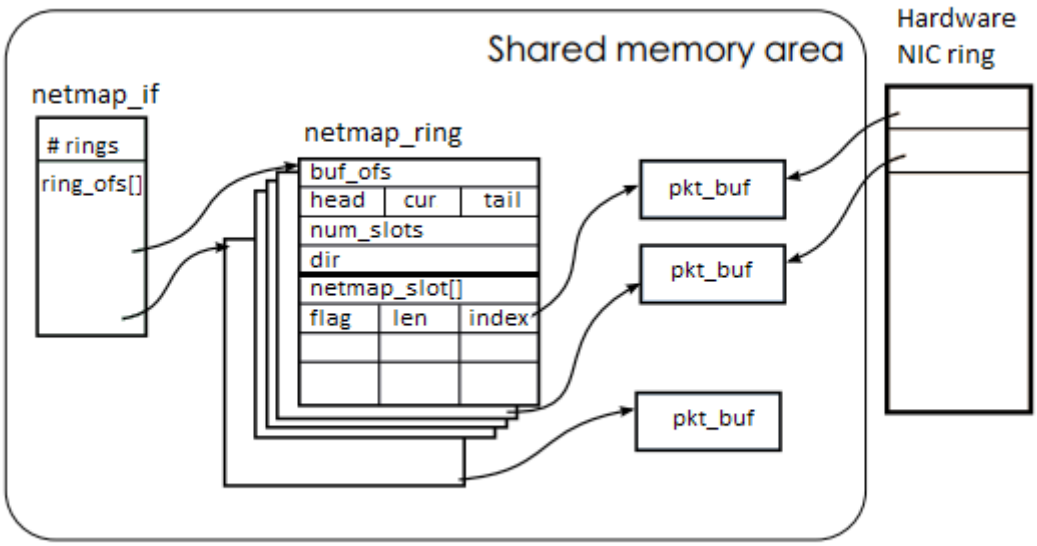


Figure 1.1: *The Netmap internal architecture*

Using this structures and a set of macros provided by the framework to directly access them, a user-space application can queue packets or grab them from a memory area shared with a hardware NIC. The slots constituting the Netmap rings, in case of hardware with customized drivers, resides in the DMA area of the hardware rings so through Netmap an application can bypass all the generic purpose stack and interact directly with the NIC, speeding up the process of transmitting and receiving packets.

1.2 The Netmap API

To switch a NIC in Netmap mode, disconnect it from the OS stack and start using it with the Netmap API, a user-space program thread just needs to follow the next steps:

- open a file descriptor to /dev/netmap
- call an ioctl() to netmap indicating the device to be used
- call mmap() and get a reference to the netmap_rings
- access data in the rings by writing or reading it
- update the status of the rings calling either poll() or ioctl() dependent from the architecture of the application.

1.3 VALE switch and Netmap pipes

VALE [2] is a feature of the Netmap framework: it can be used as a way to interconnect virtual machines, as a high speed bus between processes, as a software switch or to test high speed networking apps for example.

VALE is a software switch optimized for high performances: the enhancements are reached by using multiple techniques like batching, data prefetching and skipping the switch ports for which there is no traffic.

Another important feature is the Netmap pipe mechanisms: with this mechanism two applications are directly connected and can exchange data at rates faster than the ones achievable with the switch. This is possible because the pipes implement a zero-copy mechanism: a receiver, instead of copying the data of the transmitter, exchange a pointer to an empty slot with one that contains data from the transmitter. In this way the cost of exchanging data can be considered uninfluential while considering all the aspects of data transmission and an higher level of performance can be reached.

Chapter 2

The Netmap core kernel module

The Netmap core is the kernel module that contains all the logic for the fundamentals function of the Netmap framework like, for example packet enqueuing and switching; furthermore the core module is the point of access for user-space applications that needs to use the framework.

Usually an operating system is made of two distinct spaces: a user-space where every user application run, and a kernel-space where the core structures of the operating system resides.

The kernel structures are obviously inaccessible directly from the user space but an operating system always provides the meaning to access these structures using a kernel module as an interface: because a kernel module is trusted by the operative system, it can safely manipulate all the kernel structures.

In the Windows operating system, the only way allowed to communicate with a kernel module is via a specific system call named DeviceIoControl; this call is similar to the POSIX environment IOCTL (Device Input/Output Control) and is used to send to a specific kernel module a user-defined control code, two values representing the size of the data that will be sent and received in the call and two pointers, one to an input and one to an output structure.

In this call the control code defines what kind of operation is required and two pointers define where the kernel module can read and write data from/to the user-space.

2.1 Necessary steps to make user applications and kernel modules communicate

To be called from user-space in the Windows environment, a kernel module must do the following steps.

- Register a unique name in the \Device\ and in the \DosDevices\ names root and link them together.
- Initialize the function pointers to the IRP_MJ_CREATE, IRP_MJ_CLOSE, IRP_MJ_INTERNAL_DEVICE_CONTROL and IRP_MJ_DEVICE_CONTROL inside the array of the dispatch routines. These will tell the operating system where to find the custom function that handles the selected events.
- Make these changes stable calling RegisterDevice()

The user-space counterpart must do the following steps to connect to a loaded kernel module:

- Allocate an HANDLE: this opaque structure is just a pointer used to address a device and replaces the use of a file descriptor in POSIX.
- Open the device using the unique name previously registered by the kernel module and store it in the allocated HANDLE structure: the HANDLE will be used after to make IOCTL calls.

As soon as these operation are successfully complete, a user-mode application can start to request operations to the kernel module via DeviceIoControl() calls.

2.2 IOCTL codes specifications

An IOCTL code is represented by a *LONG* variable that is a combination if four major fields: the device type, the required access, the function code and the transfer type [3]: the two fields that are most important in our case are the last two ones.

The function code is the number that specifies what operation the user application requires from the kernel module: this number is analyzed by the handler of the IRP_MJ_DEVICE_CONTROL function.

The transfer type indicates how data will be passed between user and kernel space; the Windows operating system provides three types of transfer modes:

- METHOD_BUFFERED: with this method the OS allocates a space that is the maximum between the input and the output buffers. Then it copies data back and forth in this buffer. This method is the less efficient but the most secure. It is usually used to transfer small quantities of data.
- METHOD_IN(OUT)_DIRECT: this methods are used to transfer large quantities of data. It doesn't allocates any memory but just remaps the addresses of the buffers passed from user space to kernel space address. It is the most efficient method.
- METHOD_NEITHER: this method doesn't remap any address but instead passes the user-space address unmodified: it is care of the kernel module to assure that the address is valid and accessible.

2.3 Passing data back and forth between kernel and user-space

For every IOCTL call is expensive (at least 1 microsecond per call), the Netmap original code solved this problem by mapping the kernel-space Netmap structures directly to user-space applications. In this way an application can directly read/write and issue IOCTLs only to synchronize its structures with the kernel ones when it's really needed.

In the FreeBSD/Linux world, this can be done with mmap(): this call is available in a POSIX compliant kernel-space module and can be handled directly by the module itself, including the page-fault calls; in Windows this handler unfortunately doesn't exists: this is a design choice that has been chosen to reduce security problems with kernel exposed memory space and consequently, to limit the security checks of memory addresses from

user-space programs to kernel memory..

To cope with this problem it has been decided to remap the mmap() function with an IOCTL: this call passes a structure that will contain the pointer to the remapped kernel-space memory into user-space memory.

The structure is described in netmap.h as follows:

Code listing 2.1: *MEMORY_ENTRY structure*

```
typedef struct _MEMORY_ENTRY {
    PVOID       pUsermodeVirtualAddress;
} MEMORY_ENTRY, *PMEMORY_ENTRY;
```

Inside the kernel module the kernel-space memory is mapped in a non contiguous way but in user-space this memory is seen as contiguous: to make this possible the following function has been written:

Code listing 2.2: *win32_build_user_vm_map*

```
PMDL win32_build_user_vm_map(struct netmap_mem_d* nmd)
{
   int i, j;
   u_int memsize, memflags, ofs = 0;
   PMDL mainMdl, tempMdl;
   if (netmap_mem_get_info(nmd, &memsize, &memflags, NULL)) {
     D("memory not finalised yet");
     return NULL;
   }
   mainMdl = IoAllocateMdl(NULL, memsize, FALSE, FALSE, NULL);
   if (mainMdl == NULL) {
     D("failed to allocate mdl");
     return NULL;
   }
   NMA_LOCK(nmd);
   for (i = 0; i < NETMAP_POOLS_NR; i++) {
     struct netmap_obj_pool *p = &nmd->pools[i];
     int clsz = p->_clustsize;
     int clobjs = p->_clustentries; /* objects per cluster */
     int mdl_len = sizeof(PFN_NUMBER) * BYTES_TO_PAGES(clsz);
     PPFN_NUMBER pSrc, pDst;
```

```
    /* each pool has a different cluster size so we need to reallocate */
    tempMdl = IoAllocateMdl(p->lut[0].vaddr, clsz, FALSE, FALSE, NULL);
    if (tempMdl == NULL) {
        NMA_UNLOCK(nmd);
        D("fail to allocate tempMdl");
        IoFreeMdl(mainMdl);
        return NULL;
    }
    pSrc = MmGetMdlPfnArray(tempMdl);
    /* create one entry per cluster, the lut[] has one entry per object */
    for (j = 0; j < p->numclusters; j++, ofs += clsz) {
        pDst = &MmGetMdlPfnArray(mainMdl)[BYTES_TO_PAGES(ofs)];
        MmInitializeMdl(tempMdl, p->lut[j*clobjs].vaddr, clsz);
        MmBuildMdlForNonPagedPool(tempMdl); /* compute physical page addresses */
        RtlCopyMemory(pDst, pSrc, mdl_len); /* copy the page descriptors */
        mainMdl->MdlFlags = tempMdl->MdlFlags; /* XXX what is in here ? */
    }
    IoFreeMdl(tempMdl);
  }
  NMA_UNLOCK(nmd);
  return mainMdl;
}
```

This function builds a unique Memory Descriptor List (MDL [4]) that comprehends all the various blocks of memory allocated as a buffer for a Netmap interface: after this operation this block is translated to user-space addressing using the MmMapLockedPagesSpecifyCache() function.

The result of these operations is returned as an IOCTL output in a MEMORY_ENTRY, structure.

2.3.1 Memory descriptor list (MDL)

The previous remap is possible thanks to the MDL structure: this is a partially opaque structure used to describe how a virtual memory buffer is really allocated in physical memory..

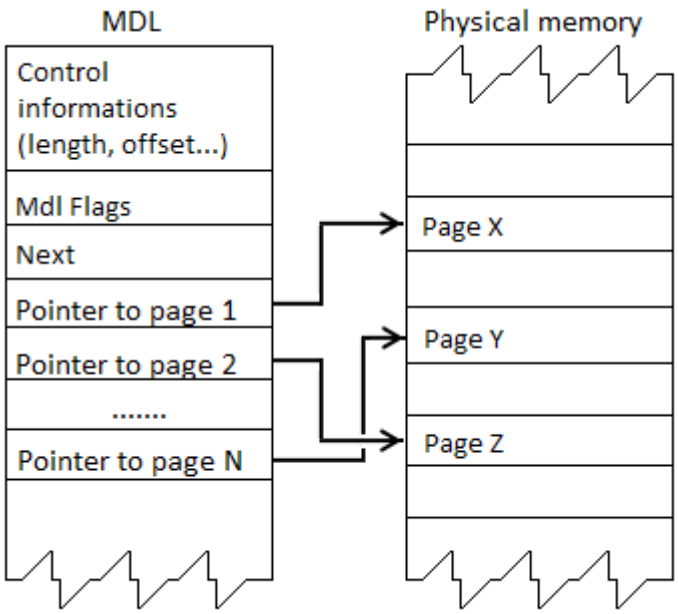


Figure 2.1: *How MDL works*

As shown in figure 1.1, an MDL contains some private information about the total size of the mapped memory, some flags, a pointer to a subsequent MDL used to create a chain of MDLs and a list of pointers to physical pages.

Using this knowledge it is possible to use the IoAllocateMdl() function to allocate an MDL structure with enough pointers to address the needed memory size: after having done that, it is possible to build an entire MDL composed by pages distributed in physical memory by computing the respective virtual address and copying the information in the main MDL.

2.4 Port of the poll mechanism

In the POSIX environment, Poll() is defined as a mechanism for multiplexing input/output over a set of file descriptors: this is useful because while a user-space application has nothing to do, can go to sleep and wait for an event to wake her up and continue to run its routines.

In the windows environment there's nothing like that to be used with a device HANDLE, so it has been chosen to emulate this mechanism with an IOCTL and a user defined structure.

***Code listing 2.1**: The POLL_REQUEST_DATA structure*

```
typedef struct _POLL_REQUEST_DATA {
    int events;   //Type of event like POLLIN or POLLOUT
    int timeout; //Selected timeout before exit the call
    int revents;  //Returned result from the kernel module
} POLL_REQUEST_DATA;
```

The structure, passed as an input and an output for the IOCTL, emulate the *pollfd* structure used in a call to Poll() in a POSIX environment: the pollfd structure is represented by the first two fields and the *revents* field represents the output from the poll original call.

The call inside the kernel module is emulated using a custom defined structure, the win_SELINFO structure: this structure is made of two objects, a KEVENT object and a KGUARDED_MUTEX object.

***Code listing 2.2:** The win_SELINFO structure*

```
typedef struct _win_SELINFO
{
    KEVENT queue;
    KGUARDED_MUTEX mutex;
} win_SELINFO;
```

When a thread issue a call to poll, the Netmap module examines its structures to understand if it's possible to cope with the request: if the request cannot be satisfied the

thread is left to sleep for as long as the timeout parameter specifies on a queue object. As soon as some other thread will send a signal on the same queue, the first thread will wake up and finish its task.

The mutex object in the structure is needed because the queue object can be accessed by different threads in the same moment and this could lead to instabilities and subsequent problems in the synchronization between threads.

Code listing 2.3: *extract from netmap_windows.c :: ioctlDeviceControl*

```
case NETMAP_POLL:
{
   POLL_REQUEST_DATA *pollData = data;
   LARGE_INTEGER tout;
   tout.QuadPart = -(int)(pollData->timeout) * 1000 * 10;
   struct netmap_priv_d *priv = irpSp->FileObject->FsContext;
   irpSp->FileObject->FsContext2 = NULL;
   pollData->revents = netmap_poll(priv, pollData->events, irpSp);
   while ((irpSp->FileObject->FsContext2 != NULL) && (pollData->revents == 0)) {
     NTSTATUS waitResult = KeWaitForSingleObject(&((win_SELINFO*)irpSp->FileObject->FsContext2)->queue,
                                    UserRequest, KernelMode,
                                    FALSE, &tout);
     if (waitResult == STATUS_TIMEOUT) {
          pollData->revents = STATUS_TIMEOUT;
          NtStatus = STATUS_TIMEOUT;
          break;
          }
          pollData->revents = netmap_poll(priv, pollData->events, irpSp);
     }
   irpSp->FileObject->FsContext2 = NULL;
   copy_to_user((void*)data, &arg, sizeof(POLL_REQUEST_DATA), Irp);
}
```

Chapter 3

Generic NIC hook, the NDIS 6 LWF kernel module

Netmap needs to be hooked to a network interface card to work in a real production environment; for this reason an NDIS 6 Light Weight Filter has been developed; this type of driver is provided by Windows's network architecture and belongs to the family of NDIS (Network Driver Interface Specification) drivers [5].

The NDIS library has been developed to create a modular paradigm for network drivers where each layer has a well defined task to perform; the NDIS layer has the task to connect the modules composing these this layers and to maintain the necessary information regarding the state and the parameters of every driver.

3.1 Windows network stack

The Windows network stack is a complex collection of different modules that interacts with each other. A simplified version that contains only a small subset of the real one, can be depicted as the one in the following figure:

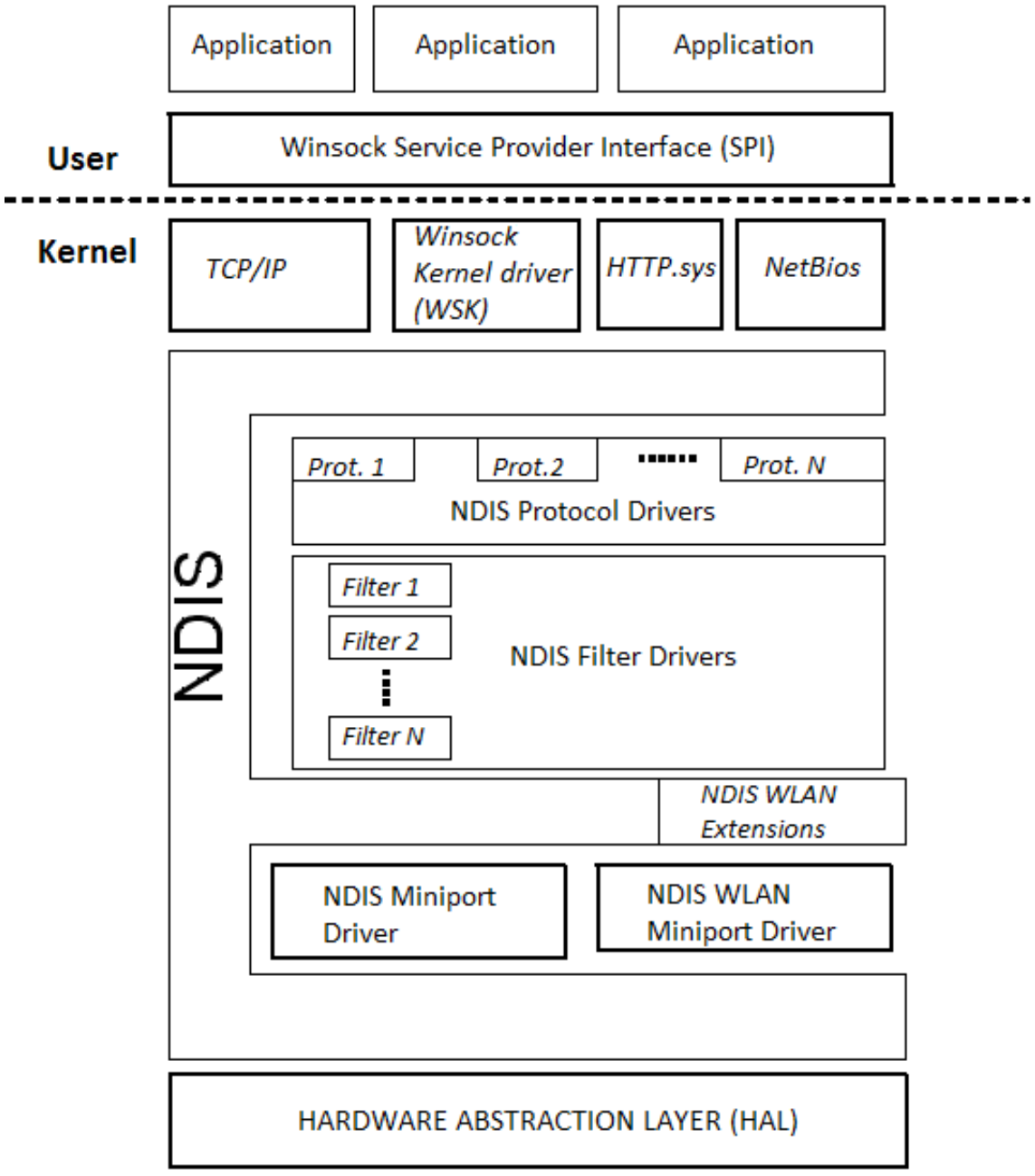


Figure 3.1: *Simplified Windows network stack*

For the objective of this work, the interesting part of the windows network stack is the NDIS portion: this part lays at the lowest level possible, before the Hardware Abstraction Layer and provides the necessary mechanisms for packet interception and injection.

The NDIS network stack provides three types of kernel module standards, every of them with a specific functionality:

- Protocol Drivers: the upper level and the nearest one to the OS; this kind of drivers are used to implement a protocol, like TCP/IP for example, or for packet monitoring
- Filter Drivers: this kind of drivers serve the purpose of filter the packets arriving from protocol drivers and from miniport drivers and can inject its own generated packets in the stack.
- Miniport Drivers: the lower level and the nearest to the hardware: this kind of drivers are tightly bound with the hardware and serves the purpose of translating the data received by the hardware into a standard packet encapsulation and in the other sense.

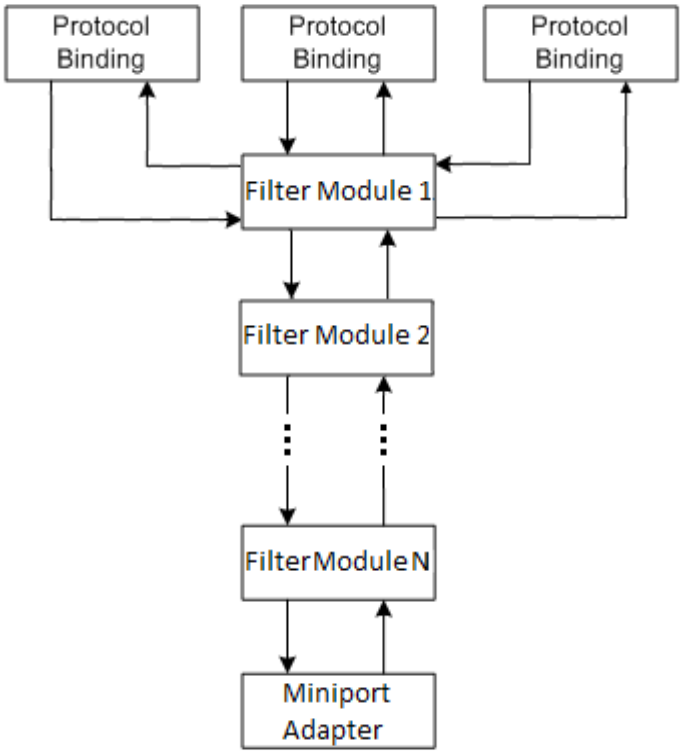


Figure 3.2: *The NDIS network layer*

3.2 Network packet data representation in NDIS 6

All of the types of module described in the previous section, works with a common data structure.

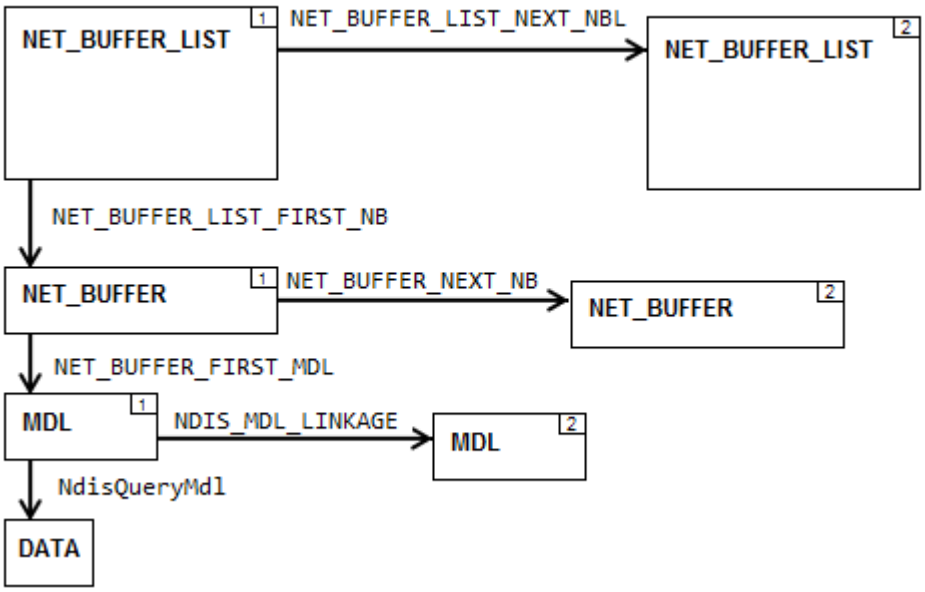


Figure 3.3: *NDIS data representation*

In this structure, a NET_BUFFER_LIST describes a list of packets that have common out of band data, a NET_BUFFER describes a single network packet and an MDL describes a single virtually contiguous data buffer. [6]

The arrows in the figure represents the macros to obtain the pointer to the linked structure with the exception of the NdisQueryMdl() function.

The pointer to the first NET_BUFFER_LIST is what is exchanged between the various NDIS module levels.

A NetBufferList and all the linked structures are property of the module who has generated them, so a copy of the packet is needed before indicating that the module has done with the data: this is needed to warn the owner of the packet can safely free the memory allocated for the NET_BUFFER_LIST structure and all the other linked structures..

In the case of packets injected by the Netmap NDIS module, all the structures allocated

for the packets must be cleared when the callback functions SendNetBufferListsCompleteHandler() and ReturnNetBufferListsHandler() are issued from other drivers.

3.3 NDIS6 LWF module

To implement this kernel module, the 'NDIS 6.0 Filter' example from the WDK 8.1 library has been used as base: during the develop of the module, the set of added functions needed for the interaction with the core module, have been kept as compact as possible to minimize the differences between the original code and the final one.

Regarding the architecture, an NDIS6 kernel module is very similar to a classic WDM driver: this kind of driver must implement a DriverEntry function, where a unique name is declared and the pointers to the needed filter functions are hooked to custom implementations.

The major functions pointer handlers are stored in a structure called NDIS_FILTER_DRIVER_CHARACTERISTICS [7]; the most useful pointers in the developed driver are the following:

- AttachHandler: used by the OS during the install phase, tries to attach the filter to an interface: it also is used to build the necessary structures to catch and inject packets from/to an interface.
- DetachHandler: used by the OS during the uninstall phase, it detaches the filter from an interface and clear the memory allocated during the Attach call.
- SendNetBufferListsHandler: used to intercept the packets incoming from the OS; called from a higher level driver when the OS needs to send packets.
- SendNetBufferListsCompleteHandler: the asynchronous callback called from a lower level driver when it has completed its manipulation of a group of packets; needed for cleanup operations.
- ReceiveNetBufferListsHandler: used to intercept the packets incoming from a NIC, called from a lower level driver when it has received new packets.
- ReturnNetBufferListsHandler: the asynchronous callback called from a higher

level driver when it has completed it's manipulation of a group of packets; needed for cleanup operations.

Other than this, in the DriverEntry function has been implemented a mechanism to communicate with the Netmap core module: this mechanism is build as an inter-driver IOCTL and is received by the Netmap core module in the handler of the IRP_MJ_INTERNAL_DEVICE_CONTROL: the reason that had driven this choice is that using this major function, this kind of request can only be issued from a module located in kernel-space and not directly by an application in user-space.

This call check for the presence in the system of a loaded Netmap core and, in case of success, exchange the pointers to a set of functions with the core module.

Code listing 3.1: *extract from nm-ndis DriverEntry function*

```
OBJECT_ATTRIBUTES   attr;
UNICODE_STRING      name;
IO_STATUS_BLOCK     iosb;
PIRP pIrp;

RtlInitUnicodeString(&name, NETMAP_DOS_DEVICE_NAME);
InitializeObjectAttributes(&attr, &name, OBJ_CASE_INSENSITIVE | OBJ_KERNEL_HANDLE, NULL, NULL);
Status = IoGetDeviceObjectPointer(&name, FILE_ALL_ACCESS, &netmap_fp, &netmap_dev);
if (Status != NDIS_STATUS_SUCCESS) {
   DEBUGP(DL_WARN, "Cannot find netmap driver\n");
   break;
}
// prepare input parameters returned by the netmap ioctl
netmap_hooks.handle_tx = NULL;
netmap_hooks.handle_rx = NULL;
// and output parameters that we pass to it
netmap_hooks.ndis_regif = &ndis_regif;
netmap_hooks.ndis_rele = &ndis_rele;
netmap_hooks.injectPacket = &injectPacket;

pIrp = IoBuildDeviceIoControlRequest(NETMAP_KERNEL_XCHANGE_POINTERS, netmap_dev,
            &netmap_hooks, sizeof(FUNCTION_POINTER_XCHANGE),
            &netmap_hooks, sizeof(FUNCTION_POINTER_XCHANGE),
            TRUE,
            NULL,
            &iosb);
```

```
if (pIrp != NULL) {
   Status = IoCallDriver(netmap_dev, pIrp);
} else {
   Status = STATUS_DEVICE_DOES_NOT_EXIST;
}
```

This has been done to speed up the communication between the two modules; for example when the NDIS filter receives packets from the underlying miniport driver, can send the data to the Netmap core by calling the netmap_hooks.handle_rx() function. At the same time, if the Netmap core needs to send a packet, can do it by calling the inject_packet() function shared by the NDIS module.

3.4 The FUNCTION_POINTER_XCHANGE structure

The FUNCTION_POINTER_XCHANGE is a structure defined in win_glue.h and created to pass function pointers between the Netmap core module and the NDIS module; is defined as follows:

Code listing 3.2: *The FUNCTION_POINTER_EXCHANGE structure*

```
typedef struct _FUNCTION_POINTER_XCHANGE {
    /* ndis -> netmap calls */
    struct NET_BUFFER* (*handle_rx)(struct net_device*, uint32_t length, const char* data);
    struct NET_BUFFER* (*handle_tx)(struct net_device*, uint32_t length, const char* data);
    /* netmap -> ndis calls */
    NTSTATUS (*ndis_regif)(struct net_device *ifp);
    NTSTATUS (*ndis_rele)(struct net_device *ifp);
    PVOID (*injectPacket)(PVOID _pfilter, PVOID data, uint32_t length, BOOLEAN sendToMiniport,
PNET_BUFFER_LIST prev);
} FUNCTION_POINTER_XCHANGE;
```

As the call described in the previous paragraph is successfully completed the Netmap core will have the direct kernel address pointers to the ndis_regif, ndis_rele and injectPacket functions and the NDIS module will have the pointers to the handle_rx and handle_tx functions.

The exported functions are mapped as follows:

Function pointer	Used function
handle_rx	netmap_windows.c :: windows_handle_rx()
handle_tx	netmap_windows.c :: windows_handle_tx()
ndis_regif	device.c :: ndis_regif()
ndis_rele	device.c :: ndis_rele()
injectPacket	device.c :: injectPacket()

The only drawbacks of this architecture is that before every call the calling module must assure that the pointer is not NULL and that whenever one of the two modules is unloaded in its unloading routine the pointer for the exposed functions must be set to NULL.

3.5 Attaching to an interface

When an NDIS driver is installed, the first routine called by the operating system is the DriverEntry(); after this call, for every compatible interface present on the system, the OS issue a call to the AttachHandler pointed function specifying the interface at every call.

This function must allocate and initialize data structures for a passed filter instance: this structures basically comprehends the name of the interface, the Interface Index (IfIndex), and other common attributes.

What has been added in this function other than the original code, is the allocation of a special pool used to create the NetBufferLists and the NetBuffers and the allocation of a Non Paged Look Aside Buffer used to create data buffers: the pool and the LookAsideBuffer are used to inject packets into the stack.

As a last operation, a flag to set if either capture or let packets pass untouched is set to permit the second option.

3.6 Packets capture

In the NDIS filter, the packet capture is done by implementing two functions and hooking them up to their respective function pointers.

To intercept packets coming from the operating system, is needed to implement the SendNetBufferListsHandler major function; to intercept packets coming from the network card, is needed to implement the ReceiveNetBufferListsHandler() major function.

These functions are quite similar, the only differences are the Netmap core functions to be called as soon as a packet arrives and obviously the functions to indicate to other drivers that the module has done manipulating the packets.

For the SendNetBufferListsHandler as soon as all the packets linked to the incoming NetBufferList are copied to a Netmap queue, the function issue the call NdisFSendNetBufferListsComplete(): with this call the packets are not passed to the next level driver but the source driver is informed that it can safely clean its structures. In the case the module and the interface that has received the packets is not already attached to Netmap core, calling NdisFSendNetBufferLists() the module just pass the data to the next level driver.

Code listing 3,4: *Extract from nm-ndis FilterSendNetBufferLists Function*

```
if (netmap_hooks.handle_tx != NULL && (pFilter->intercept & NM_WIN_CATCH_TX)) {
   int result = -1;
   PNET_BUFFER pkt = NULL;
   PNET_BUFFER_LIST current_list = NetBufferLists;
   while (current_list && (pkt || NULL != (pkt = NET_BUFFER_LIST_FIRST_NB(current_list)))) {
      PVOID buffer = NdisGetDataBuffer(pkt, pkt->DataLength, NULL, 1, 0);
      if (buffer != NULL)
            result = netmap_hooks.handle_tx(pFilter->ifp, pkt->DataLength, buffer);
      pkt = pkt->Next;
      if (pkt == NULL)
            current_list = NET_BUFFER_LIST_NEXT_NBL(current_list);
      NdisFSendNetBufferListsComplete(pFilter->FilterHandle, NetBufferLists, SendFlags);
}  else {
         NdisFSendNetBufferLists(pFilter->FilterHandle, NetBufferLists, PortNumber, SendFlags);
}
```

The same mechanism is applied to the ReceiveNetBufferListsHandler with the difference that the function call to indicate the previous level driver to safely deallocate the data buffers is NdisFReturnNetBufferLists() and the bypass function is NdisFIndicateReceiveNetBufferLists() as shown in the following snippet.

Code listing 3.5: *Extract from nm-ndis FilterSendNetBufferLists Function*

```
if (netmap_hooks.handle_rx != NULL && (pFilter->intercept & NM_WIN_CATCH_RX)) {
   int result = -1;
   PNET_BUFFER pkt = NULL;
   PNET_BUFFER_LIST current_list = NetBufferLists;

   while (current_list && (pkt || NULL != (pkt = NET_BUFFER_LIST_FIRST_NB(current_list)) ) ) {
      PVOID buffer = NdisGetDataBuffer(pkt, pkt->DataLength, NULL, 1, 0);
      if (buffer != NULL)
            result = netmap_hooks.handle_rx(pFilter->ifp, pkt->DataLength, buffer);
      pkt = pkt->Next;
      if (pkt == NULL)
     current_list = NET_BUFFER_LIST_NEXT_NBL(current_list);
   }
   NdisFReturnNetBufferLists(pFilter->FilterHandle, NetBufferLists, ReceiveFlags);
} else {
   NdisFIndicateReceiveNetBufferLists( pFilter->FilterHandle,
                  NetBufferLists,
                  PortNumber,
                  NumberOfNetBufferLists,
                  ReceiveFlags);
}
```

3.7 Packets injection

To make the injection of packets possible, in the NDIS module the injectPacket function has been implemented.

Code listing 3.6: *injectPacket function of nm-ndis driver*

```
PVOID injectPacket(PVOID _pfilter, PVOID data, uint32_t length, BOOLEAN sendToMiniport,
PNET_BUFFER_LIST prev)
{
```

```
PVOID              buffer = NULL;
PMDL                pMdl = NULL;
PNET_BUFFER_LIST        pBufList = NULL;
PNET_BUFFER            pFirst = NULL;
PVOID              pNdisPacketMemory = NULL;
NTSTATUS            status = STATUS_SUCCESS;
PMS_FILTER              pfilter = (PMS_FILTER)_pfilter;

do {
        if (data == NULL && prev != NULL) {
    pBufList = prev;
    goto sendOut;
  }
  if (data == NULL)
    return NULL;
  buffer = ExAllocateFromNPagedLookasideList(&pfilter->netmap_injected_packets_pool);
  if (buffer == NULL) {
    DbgPrint("Error allocating buffer!\n");
    status = STATUS_INSUFFICIENT_RESOURCES;
    break;
  }
  pMdl = NdisAllocateMdl(pfilter->FilterHandle, buffer, length);
  if (pMdl == NULL) {
    DbgPrint("nmNdis.sys: Error allocating MDL!\n");
    status = STATUS_INSUFFICIENT_RESOURCES;
    break;
  }
  pMdl->Next = NULL;
  pBufList = NdisAllocateNetBufferAndNetBufferList(pfilter->netmap_pool,
          0, 0,
          pMdl, 0,
          length);
  if (pBufList == NULL) {
    DbgPrint("nmNdis.sys: Error allocating NdisAllocateNetBufferAndNetBufferList!\n");
    status = STATUS_INSUFFICIENT_RESOURCES;
    break;
  }
  pFirst = NET_BUFFER_LIST_FIRST_NB(pBufList);
  pNdisPacketMemory = NdisGetDataBuffer(pFirst, length, NULL, sizeof(UINT8), 0);
  if (pNdisPacketMemory == NULL) {
    DbgPrint("nmNdis.sys: weird, bad pNdisPacketMemory!\n");
    status = STATUS_INSUFFICIENT_RESOURCES;
```

```
        break;
    }
    NdisMoveMemory(pNdisPacketMemory, data, length);
    pBufList->SourceHandle = pfilter->FilterHandle;
    if (prev != NULL) {
        prev->Next = pBufList;
    }
    return pBufList;
sendOut:
    if (sendToMiniport) {
            // This send down to the NIC (miniport)
        NdisFSendNetBufferLists(pfilter->FilterHandle, pBufList, NDIS_DEFAULT_PORT_NUMBER, 0);
    } else {
            // This one sends up to the OS
        int nblNumber = 1;
        PNET_BUFFER_LIST temp = prev;
        while (temp->Next != NULL) {
            temp = temp->Next;
            nblNumber++;
        }
            NdisFIndicateReceiveNetBufferLists(pfilter->FilterHandle, pBufList, NDIS_DEFAULT_PORT_NUMBER,
nblNumber, 0);
    }
  } while (FALSE);
  if (status != STATUS_SUCCESS) {
    if (pBufList)
        NdisFreeNetBufferList(pBufList);
    if (pMdl)
        NdisFreeMdl(pMdl);
    if (buffer)
        ExFreeToNPagedLookasideList(&pfilter->netmap_injected_packets_pool, buffer);
    return NULL;
  }
  return pBufList;
}
```

This function is directly called by the Netmap core module and it has five parameters:

- PVOID pFilter: pointer to the MSFILTER that describes the network interface where is needed to inject a list of packets
- PVOID data: pointer to the data of the packet to be injected

- uint32_t length: length of the packet to be injected
- BOOLEAN sendToMiniport: indicates to either send to the NIC or to the operating system
- PNET_BUFFER_LIST prev: used to attach the currently passed packet to a list

This function has two behaviors: if data differs from NULL the procedure allocate a buffer, a NetBuffer and a NetBufferList from the pools previously allocated in the FilterAttach procedure, then enqueues it in the list passed in the prev pointer; the function returns the newly generated packet pointer.

On the other hand, if data is equal to NULL the function interprets the prev pointer as a ready batch to be transmitted: in this case the function examines the BOOLEAN value of the variable sendToMiniport to decide where the packets are needed to be sent, and then send the batch of packets either to the NIC or to the operating system.

As soon as a chain of drivers has done the operations on the packets, a callback is issued to the Netmap NDIS module: if the packets have been sent to the miniport a call to SendNetBufferListsCompleteHandler() will be issued or, if the packets have been sent to the operating system, a call to ReturnNetBufferListsHandler() will be issued.

In these two calls the Netmap module must release all the structures it has allocated to inject the packets: the modifications to the original functions are the following:

Code listing 3.7: *Extract from the FilterSendNetBufferListComplete function*

```
PNET_BUFFER_LIST pCurrNBL = NetBufferLists;
  while (pCurrNBL != NULL) {
       PNET_BUFFER_LIST pNextNBL = NET_BUFFER_LIST_NEXT_NBL(pCurrNBL);
       NET_BUFFER_LIST_NEXT_NBL(pCurrNBL) = NULL;
       if (pCurrNBL->NdisPoolHandle == pFilter->netmap_pool) {
    PNET_BUFFER pCurrNB = NET_BUFFER_LIST_FIRST_NB(pCurrNBL);
    while( pCurrNB != NULL ) {
        PNET_BUFFER pNextNB = NET_BUFFER_NEXT_NB(pCurrNB);
        PMDL pCurrMDL = NET_BUFFER_FIRST_MDL(pCurrNB);
        while( pCurrMDL != NULL ) {
            PVOID pDataBuffer = NULL;
            uint32_t ulDataLength = 0;
            PMDL pNextMDL = NDIS_MDL_LINKAGE(pCurrMDL);
            NdisQueryMdl(pCurrMDL, (PVOID *)&pDataBuffer, &ulDataLength, NormalPagePriority);
            NdisFreeMdl(pCurrMDL);
```

```
            if( pDataBuffer != NULL ) {
                ExFreeToNPagedLookasideList(&pFilter->netmap_injected_packets_pool, pDataBuffer);
        }
            pCurrMDL = pNextMDL;
    }
    pCurrNB = pNextNB;
     }
     NdisFreeNetBufferList(pCurrNBL);
     } else {
     NdisFSendNetBufferListsComplete(pFilter->FilterHandle, pCurrNBL, SendCompleteFlags);
    }
pCurrNBL = pNextNBL;
}
```

This block is a part of the FilterSendNetBufferListsComplete() function and its task is to cycle all the NetBufferLists and clear all the underlying structures.

When the filter driver receives this callback, it has no guarantees that the packets have really been sent: the only thing that can be inferred by this call is that the next driver has done manipulating the data structures so the NDIS filter driver can run the clean-up operations.

In the FilterReturnNetBufferLists() happens the same as the block that has been examined above: the only difference is the NdisFSendNetBufferListsComplete() function call that is substituted by a call to NdisFReturnNetBufferLists().

3.8 Driver unload and clean-up

When the driver is unloaded the function hooked to the FilterDetach for every interface and finally FilterUnload are called.

The first function is used to detach the module from an interface network stack: the following snippet is what has been added to the original code to clean up the custom structures that have been allocated during the FilterAttach procedure.

Code listing 3.8: *Extract from FilterAttach procedure*

```
 if (pFilter->netmap_pool != NULL) {
       NdisFreeNetBufferListPool(pFilter->netmap_pool);
       pFilter->netmap_pool = NULL;
       NdisZeroMemory(&pFilter->PoolParameters, sizeof(NET_BUFFER_LIST_POOL_PARAMETERS));
}
if (&pFilter->netmap_injected_packets_pool != NULL)
{
   ExDeleteNPagedLookasideList(&pFilter->netmap_injected_packets_pool);
}
```

The FilterUnload routine originally contained only the code needed to deregister the filter and remove the name of the filter from the roots .\Device and .\DosDevices .

The modification that has been done added an IOCTL call to the Netmap core driver, used to inform the destination driver that the NDIS filter will be no longer available. Essentially this code sets the various shared pointers to NULL so the core module won't be able to call these functions anymore.

Code listing 3.9: *Extract from FilterUnload procedure*

```
if (netmap_fp != NULL) {
   PIRP pIrp;
   NTSTATUS Status;
   IO_STATUS_BLOCK iosb;
   // prepare input parameters returned by the netmap ioctl
   netmap_hooks.handle_tx = NULL;
   netmap_hooks.handle_rx = NULL;
   // tells netmap module we are unloading
   netmap_hooks.ndis_regif = NULL;
   netmap_hooks.ndis_rele = NULL;
   netmap_hooks.injectPacket = NULL;
   pIrp = IoBuildDeviceIoControlRequest(NETMAP_KERNEL_XCHANGE_POINTERS, netmap_dev,
               &netmap_hooks, sizeof(FUNCTION_POINTER_XCHANGE),
               &netmap_hooks, sizeof(FUNCTION_POINTER_XCHANGE),
               TRUE,
               NULL,
               &iosb);
   if (pIrp != NULL)  {
     Status = IoCallDriver(netmap_dev, pIrp);
   } else {
```

```
    Status = STATUS_DEVICE_DOES_NOT_EXIST;
  }
ObDereferenceObject(netmap_fp);
netmap_fp = NULL;
netmap_dev = NULL;
}
```

Chapter 4

Porting of kernel structures and routines

For Netmap uses many routines provided by the host operating systems, critical structures, primitives and routines have been ported from the ones available under FreeBSD to their analogue objects and functions that come up with the Windows operating system. Other than the redefinition of basic types like int32_t or u uint32_t available under other names, this work needed the port of dynamic memory allocation routines and some synchronization primitives.

4.1 Dynamic memory allocation

In this work, it has been needed the use two types of dynamic memory allocation: long term and short term. For long term are considered the structures describing the interfaces, the internal structures of the kernel modules etc... in the short term case falls the structures needed for packet injection or packet interception.

For every long term allocation the ExAllocatedPoolwithTag() [8] function have been used: this routine needs to know in which kind of pool the memory will be allocated, the size of the memory and a four letter tag. In every case the type of allocation has been chosen as NonPagedPool because perform better than other kind of pools considering that

this kind of memory won't ever be paged out.

The tag parameter is an help in debug phase: with specific instruments is possible to inspect the allocated pools and detect memory leaks.

To free the memory allocated with this routine, a call to ExFreePoolWithTag() is needed.

The two functions that have been just discussed are the counterparts of malloc and free in POSIX environment.

In the case of short term memory a structure called LookAside List [9] has been used: this kind of pool has a nice particularity that lead it to be a perfect fit for fast allocations and deallocations. A LookAside List is a list of reusable and reserved fixed-size buffers; every time a request to allocate a new buffer is issued, the operating system provide to the caller one of the free entries: when the caller has done it must release the buffer to make it available again for a subsequent call. The operating system dynamically increment or decrease the number of available entries by tracking the demand for allocations and deallocations.

A lookaside list needs to be firstly initialized with the routine ExInitializeNPagedLookasideList(): this routine gets as input parameters the pointer to the lookaside list to be initialized, the size of the buffers and a tag for debug purposes.

As soon as the LAL is initialized, an application can start to request buffers using the ExAllocateFromNPagedLookasideList() and must release them with the ExFreeToNPagedLookasideList() routine.

When this kind of structure is not needed anymore, it must be destroyed by calling the ExDeleteNPagedLookasideList(). Forgetting the last step is a serious error that can lead to various problems in memory.

4.2 Mutexes, spin-locks and read/write locks

In this port, three synchronization objects has been encountered: mutexes, spin-locks and read/write locks.

Before talking about the objects just listed, a little summary on how those kind of synchronization objects work is needed: almost all synchronization objects on Windows relies on the IRQL or Interrupt Request Level [10] to be fast. Every processor has a current IRQL associated that describes the level of privilege a thread must have to make preemption on the current running thread. Raising the interrupt level is equal to mask the interrupts belonging to a lower level.

The standard IRQL are the following, listed from the lowest to the highest priority:

- PASSIVE_LEVEL
- APC_LEVEL
- DISPATCH_LEVEL
- DIRQL

By raising the IRQL, a thread in a critical section is hardly interrupted by another thread, making this architecture reach good performances.

The mutex synchronization object, implemented by the KGUARDED_MUTEX opaque structure, is the fastest available mutex object in Windows. This object must be initialized before being used by calling the KeInitializeGuardedMutex() procedure on it: after this operation, a thread can try to obtain the access to a critical section guarded by this kind of mutex by invoking KeAcquireGuardedMutex() and must release it by calling KeReleaseGuardedMutex() as soon as possible.

Another object used in the original Netmap module is the spin-lock; to port this kind of object a structure that contains the spin-lock itself and a reference to an IRQL has been needed.

Code listing 4.1: *The win_spinlock_t structure*

```
typedef struct {
    KSPIN_LOCK    sl;
    KIRQL         irql;
} win_spinlock_t;
```

This structure is a must because when a thread want to access a spin-lock guarded section, the current IRQL must be saved: in this way, when the spin-lock will be released the old IRQL can be restored at its original state. The functions to initialize, acquire and release a spin-lock are the KeInitializeSpinLock(), KeAcquireSpinLock() and KeReleaseSpinLock().

The last synchronization object that has been ported is the read/write lock: this type of synchronization primitive is used to allow concurrent access for read-only operations and give exclusive access for writing operations; read/write locks are widely used to protect data structures that shouldn't be used until a complete write of the entire structure is done.

Defined in FreeBSD as rwlock and in Linux as rw_semaphore, this object find its counterpart in Windows as ERESOURCE or Executive Resources [11]. Because this object mostly works as its POSIX counterparts, the port can be explained with a simple table.

FreeBSD	Linux	Windows
rw_init_flags	init_rwsem	ExInitializeResourceLite
rw_destroy		ExDeleteResourceLite
rw_wlock	down_write	ExAcquireResourceExclusiveLite(...,TRUE)
rw_wunlock	up_write	ExReleaseResourceLite
rw_rlock	down_read	ExAcquireResourceSharedLite
rw_runlock	up_read	ExReleaseResourceLite
rw_try_rlock	down_read_trylock	ExAcquireResourceExclusiveLite(...,FALSE)

Chapter 5

User-space applications

With the port of the kernel module, the packet-generator (pkt-gen) application has also been ported and an utility application (Loader) has been developed from scratch.

5.1 Packet-gen

Packet-generator is a user-space application developed to test the performances of a network link and the performances in general of Netmap.

This application has been useful to test the performances of the ported code to Windows in respect to the values that can be reached on FreeBSD.

As for the kernel modules, the modifications on the original code have been limited to the minimum necessary; in fact, in the pkt-gen.c source it has been needed to add only a limited amount of code.

- Implement ether_aton and ether_ntoa, two functions necessary to convert and ASCII representation of an ethernet address to binary form and in the opposite sense.
- Remap pthread_setaffinity_np to Windows SetThreadAffinityMask(), useful to pin a thread on a core.
- Remove the “sys/sysctl.h” header and add the FreeBSD header “net/if_dl.h” and “net/ethernet.h” containing structures unknown to Windows and Cygwin.

Most of the porting work has been done in the netmap_user.h header, where the custom definitions of ioctl(), mmap() and poll() are stored.

5.2 netmap_user.h header modifications

The netmap_user.h header keeps the definitions and the structures used by only the user-mode applications: so in this file has been added a Windows only section that contains the layer of compatibility that Cygwin cannot provide.

The first things added are two static variables, modified by nm_open and nm_close procedures: for Windows only, these functions have the additional task to keep a list of file descriptors and their linked Windows handle to opened Netmap core device drivers.

Keeping track of these file descriptors is useful because the user-space application can issue a call to native mmap()/ioctl()/poll() if the fd specified in the call is not present in the list: otherwise these calls are redirected to specific custom implementations of the original calls.

In this file is possible to see clearly that every interaction from user-space to kernel-space is made of I/O Controls: the mmap() and poll() remapped functions calls win_nm_ioctl_internal() with a specific IOCTL code.

The win_nm_ioctl is the core of the communication from user-space to kernel-space: it decides if the call is directed to the Netmap device or to another device and calls the right function accordingly.

In the case that a Netmap device is the destination of the ioctl, the size of the passed structures are initialized and then a call to DeviceIoControl() is made.

The input parameters and the return values of the remapped functions are compatible with the original methods.

5.3 Netmap loader utility

Netmap loader (nm-loader) is a utility program that can be used to load dynamically the Netmap core module: using this utility it is possible to not install the kernel module and so to not be started at boot time.

The main use of this kind of utility is to test the netmap kernel module on systems that doesn't need to have the module always loaded: for example it can be used to make a performance test on a different systems and decide which one fits better for a particular application.

The hot install procedure relies on a small number of routines provided by the Windows operating system [12]:

- CreateFile: used to find the driver file
- OpenSCManager: used to connect to the Service Control Manager and open the Services database
- CreateService: needed to create a new service object for a standalone driver
- OpenService/StartService: used to start the execution of a driver
- ControlService: used to stop a driver/service
- DeleteService: used to delete a service/driver from the Service Control Manager database

These API are provided by the SCM or Service Control Manager: it is a remote procedure call server and it is responsible for all the operations concerning drivers and services.

The complete code for this routine con be found in the main.c source file of the loader project but the necessary steps can be summarized as follows.

5.3.1 Hot install procedure

To install a driver by using these routines the necessary steps are the following: first find the .sys file of the module and open an instance with CreateFile(): the next step needs to open a connection with the Service Control Manager (SCM) using OpenSCManager().

In the third step is needed to create a new service object and saving it in the SCM database through a call to CreateService(); the last step is to open an instance to the newly created entry in the SCM database calling the function OpenService() and then calling StartService() specifying the handle that OpenService has returned.

5.3.2 Hot uninstall procedure

Consequently to uninstall a driver with these routines is needed to open an instance to the SCM database, open the service that is needed to be uninstalled, stop the service using ControlService() and specifying SERVICE_CONTROL_STOP as the second parameter. As soon as the call returns it is possible to remove the driver from the SCM database with the routine DeleteService().

Chapter 6

Developing environment and build output

All the port described has been developed using Visual Studio 2013 and the Windows Drivers Kit 8.1: using this pair of instruments, at the moment of writing of this thesis, is the best way to implement a new WDM driver and, in particular, is the only way to implement an NDIS 6 driver.

6.1 Output of the build process

Two ways are available to build the kernel modules: directly by the Visual Studio GUI or by using the command prompt MSBuild application: keeping in mind the last opportunity, an ad-hoc *makefile* has been built to make possible to build the kernel modules and the user-space example programs directly from a Cygwin terminal window.

The Cygwin installation must include the GCC compiler and the GNU make utility to be able to compile the entire solution.

The output of a build of the entire solution comprehends the following objects:

- Netmap core module: a folder with three files, netmap.sys, netmap.inf and netmap.cat
- Netmap NDIS module: a folder with three files, nm-ndis.sys, nm-ndis.inf and nm-ndis.cat
- Loader: the executable nm-loader.exe

- pkt-gen examples: two executable, pkt-gen.exe and pkt-gen-b.exe, that can be found under the \example directory.
- Netmap core and NDIS certificates: the certificate files netmap-pkg.cer and nm-ndis-pkg.cer; these two files contains the same information and are used to allow the installation of the modules under a test-signed enabled 64bits operating system.

6.2 INF Files

To be able to build the kernel-spaces modules, every project needs an additional INF file that contains the information about the nature of the module [13]: this kind of file is even used to install the modules.

6.2.1 netmap.inf

The most useful sections of the netmap.inf that are needed to be able to install a kernel module are the following:

- [Version]: Contains the base information about the type of driver, the version and the GUID.
- [SourceDisksFiles]: defines which files will be copied during the install procedure.
- [DestinationDirs]: the destination where the netmap.sys file will be copied: the numberic value represent one of a list of standard values described in [14]
- [DefaultInstall]: the section that will be called by the installer and specifies what must be done in the install phase: in the Netmap case just copy one file.
- [DefaultInstall.Services]: called by the installer during the install phase, it is used to tell the installer to bring up a service connected to the driver: the service specifications are defined in a custom section, named in this case [Netmap_Service_Inst].
- [Install.Remove.Services]: tells the uninstaller what to do: in this case is told to just delete the service with a reference to the DelService directive [15].

- [Netmap_Service_Inst]: this custom section contains the name to be shown, the service type, the start type and the error control type: for the flags the reader should refer to [15].

6.2.2 nm-ndis.inf

The nm-ndis.inf is a file modified from the original version provided in the “NDIS6 Filter” example from Microsoft.

The section described in the previous paragraph are valid for this INF file so will be excluded from the explanation: instead the important section in this case, the [Inst_Ndi] section, will be covered.

This section contains information dependent from the type of driver: the most important subsections in the case of the nm-ndis kernel module are the following.

- HKR, Ndi,Service,,"NetmapNdis": this subsection defines the name of the service that will be shown..
- HKR, Ndi,FilterType,0x00010001,2: this subsection defines the module as a “modifying filter”: another valid value is 'HKR, Ndi,FilterType,0x00010001, 1' for a monitoring only filter.
- HKR, Ndi\Interfaces, FilterMediaTypes,,"ethernet": this subsection defines what kind of interfaces the nm-ndis filter can attach to. The only value used is 'ethernet' for our scopes, but wan, ppip, and wlan are others valid values available.

Anyway most of the parameters are briefly described directly into the INF file or for a further extended explanation the reader should refer to [16].

6.3 Digital module signature

With the restrictions imposed on the latest Windows operating systems, mostly on 64bits versions, another option must be used to enable the built kernel modules to be loaded in the kernel-space.

These restrictions have been imposed by Microsoft to limit security problems like malware installations and to have a secure source verification: when a module is in kernel-space it can access every kind of sensible data, belonging to the system or to the users, so this method is used to limit considerably this kind of information disclosure.

This constrains the developers to build their modules with the Test-Sign option enabled so the DriverSigning application is able to generate a digital signed .sys file; these files are not meant to be used in a production environment but only in a test one.

Having signed the modules is only the first step to use them on a 64bit environment, in fact two other steps are needed: install the certificate built be Visual Studio on the development machine in the trusted certification authorities root and then set the machine in test mode. These operations will be explained more in the Installation chapter.

Chapter 7

User instructions

In order to be able to use Netmap and its features, the system must be set up and the modules must be correctly loaded in kernel-space: as said in the previous chapter, the use of Netmap in a 64bit environment needs some steps to be done before the modules can be installed.

As a last remark, the Netmap core module must always be installed before the NDIS module.

7.1 Setting up the system

Because a 64bit version of Windows is enabled to let a user install only signed/authenticated modules, a way to circumvent this mechanism was needed.

Setting every Visual Studio kernel module project to be built with a test digital signature is the first step: a test digital signature is a particular type of signature recognized by Windows as special: in this way a software distributed with this kind of signature can't be installed in an immediate manner by a simple user. In fact this type of signature, as the name says, find its only purpose in a test environment for debug purposes. As soon as the product is ready, the developer needs to requests a real signature to Microsoft used to build a redistributable package targeted for end users..

As soon as the modules are signed, the operating system must be set in a state where

the module signed with a test sign certificates are recognized and accepted: this can be done using the *bcdedit* utility provided by Windows. Bcdedit is a particular utility used to set the boot options of Windows and in this case the TESTSIGNING option is the useful one.

By issuing from an administrator privileged command prompt the “bcdedit -set TESTSIGNING ON” command, after a reboot the system will be ready to accept test signed modules: if a problem is issued by the execution of this command, is very probable that is linked with the 'secure boot' hardware feature.

In recent hardware, almost every manufacturer has implemented the secure boot feature, used to prevent the modifications of operating system core functions by untrusted software: if this feature is enabled it must be disabled to install a test-signed kernel module.

The last step to be done is to install the certificate provided with the build in the visual studio output directory into the directory of trusted certificates: this can easily be done by following these steps:

- double click on the certificates
- click on “Install certificate”
- Click “next” and then select “Place all certificates in the following store”, select the “Trusted root certification authorities” option and click next.
- Now “Finish” can be clicked and the certificate is ready to be used

After the above steps are successfully competed, the machine is ready to accept the Netmap kernel modules.

7.2 Installing the modules

7.2.1 Netmap 'core' module install procedure

As explained before there are two ways available o install the Netmap core module: the first in dynamic and can be considered as 'on-demand', the second one is to install the module to be loaded automatically at boot time.

The first option can be done using the loader utility provided with the Windows Visual Studio solution: to be able to install the module the only steps to be done are:

- open a command terminal with administrative privileges
- change into the directory containing the loader executable
- executing “nm-loader l”

If everything is successful a message that confirm that the module has been correctly loaded will be shown.

To unload the module the first two steps are the same, than the command to be executed changes into “nm-loader u”

The second solution, the one that concern to install the module at boot time is even more simple. The only necessary steps are

- find the directory containing the netmap{.inf/.sys/.cat} files
- right click on the netmap.inf file and select “install”

After a reboot the module will be correctly loaded.

7.2.2 Netmap NDIS module install procedure

To install the NDIS module at the moment only one way is available and is through the Network and sharing center.

- Open Control panel, then Network and internet, then Network and sharing center.
- Click manage network connections
- Right click on any network connection and click “Properties”
- Click “Install”
- Select the “Service” entry and choose the module to be installed
- Search the folder where the nm-ndis.inf is located and select it
- After clicking next, the module will be installed on the system

To remove the module just click on the “Netmap NDIS Filter module” and click on “Uninstall”.

7.3 Using packet-gen

Packet-gen is an example program and its use is almost straightforward: it can use VALE ports, pipes and generic interfaces so it can be said that it uses almost all the features exposed by the Netmap framework.

To send a packet to an interface the "-f tx" parameter must be specified, instead to receive data the "-f rx" parameter must be specified.

To specify an interface where to send/receive packets the "- i" parameter is the one to be used and the following points describes the type of available interfaces.

- valeX:Y : specifies to attach the pkt-gen program to the switch X in the port Y.
- valeX{Y / valeX}Y : specifies to attach the pkt-gen program to the switch X in the port Y and to use a pipe.
- netmap:ethX :specifies to attach to the physical interface with IfIndex X and send/receive from/to the NIC.
- netmap:ethX^ :specifies to attach to the physical interface with IfIndex X and send/receive from/to the OS.

Other parameters are available but the most useful, at least in the performance tests, have been the following:

- -b X : create a batch of X packets and enqueue them before issuing a send.
- -R X : rate in X packets per second to be transmitted.
- -r : using the pipes, it specifies to not touch the buffers and send rubbish data.
- -D : destination MAC address: if not specified is FF:FF:FF:FF:FF:FF
- -S : source MAC: if not specified is 00:00:00:00:00:00
- -l : packet size, the standard is 60 bytes.
- -X : dump the payload of the received packets.

The IfIndex needed to use pkt-gen with real NIC and can be acquired in two ways: on systems from Windows8.1 and over, the PowerShell "Get-NetAdapter" cmdlet is available to get all the interfaces present in a system. In older systems, this command is replaced by the command prompt command "route print" that shows in the first part of its output the necessary information.

For each of the methods, the friendly name of the interface, the mac address and the IfIndex of the interfaces are shown.

Chapter 8

Conclusions

The port can be considered successful considering the results that are going to be shown in the following paragraphs: the features present in the FreeBSD and Linux counterparts are almost all functional apart from the poll() implementation that at the moment supports only one file descriptor.

8.1 Performances

A brief set of tests has been done with the modules to assure that the port has been successful: these tests have been done with the pkt-gen provided example application on a machines with an Intel i7 4790k 4,0 GHz processor (4 Cores/8 Thread) and 16Gb of RAM.

The used NICs for the tests are an Intel I218-V for 1GbE link and an Intel X520 for 10GbE link.

The tests have been made with the CPU frequency locked: in the FreeBSD case it has only been a system control to set; in the Windows environment the CPU has been locked in the BIOS, disabling C states, the Intel Speedstep® technology and the turbo feature.

The Windows version chosen for most of the tests on bare metal has been the 8.1 64 bits.

8.1.1 *Performances of the Netmap core*

To understand whether the port has been successfully implemented or not, an extensive series of tests with pkt-gen and the VALE switch has been made.

For the first series of tests the VALE software switch implemented in Netmap has been used to exchange data between two processes:

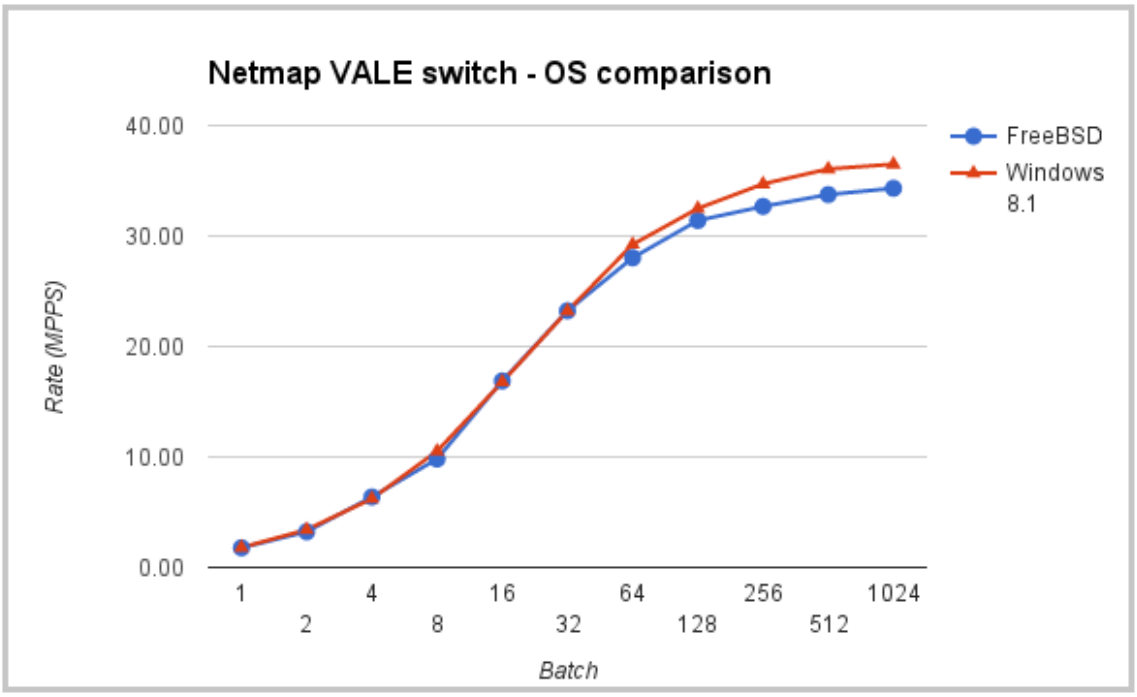
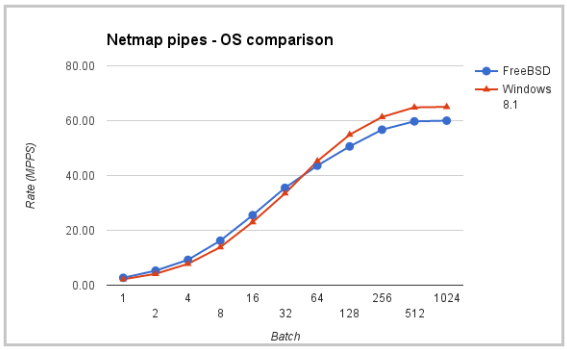
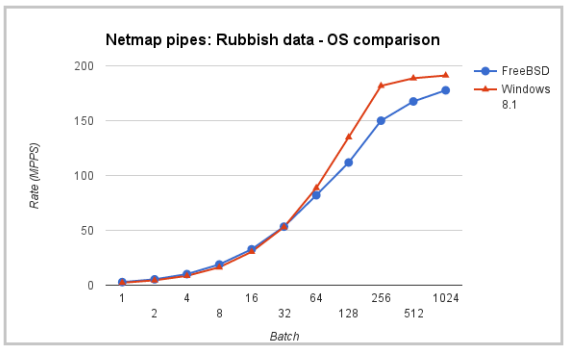


Figure 8.1: *Performances of the VALE switch*

As it is clearly visible in Figure 8.1, the results of the tests show that the performance are comparable with the ones achievable by the original code.

The second batch of tests included the utilization of the Netmap pipes feature: the first type of test has been done with exchange of real data generated by the pkt-gen application; the second one used the '-r' switch of pkt-gen to tell pkt-gen to generate data only the first time and to exchange those pre-allocated packets between the transmitter and the receiver only.





Figures 8.2 and 8.3: *Performances of the Netmap pipes in comparison*

Looking at the results shown in the figures, it is crystal clear that the performances even in this case are fully comparable to the original ones, so it is possible to say that the port of the core part of the Netmap framework can be considered successful.

8.1.2 Tests of TX/RX of data over a 1GbE link

The second batch of tests implied the use of a set of real NICs: an Intel I218-V Gigabit Ethernet and an Intel X520 working with the 10 Gigabit Ethernet standard.

The tests conducted over a machine with an Intel I218V Gigabit Ethernet NIC have demonstrated that the port to Windows can successfully reach near line rate performances in generic mode: to reach this results a few tweaks to the driver provided by Intel has been made.

The first two tweaks has been to disable the *Flow control* and the *Adaptive inter-frame space* features. With these two features disabled some tests with different values of the *Interrupt moderation* feature has been made.

For this experiment the TX rate of the machine linked to the one under test was 1,210 MPPS (millions of packets per second).

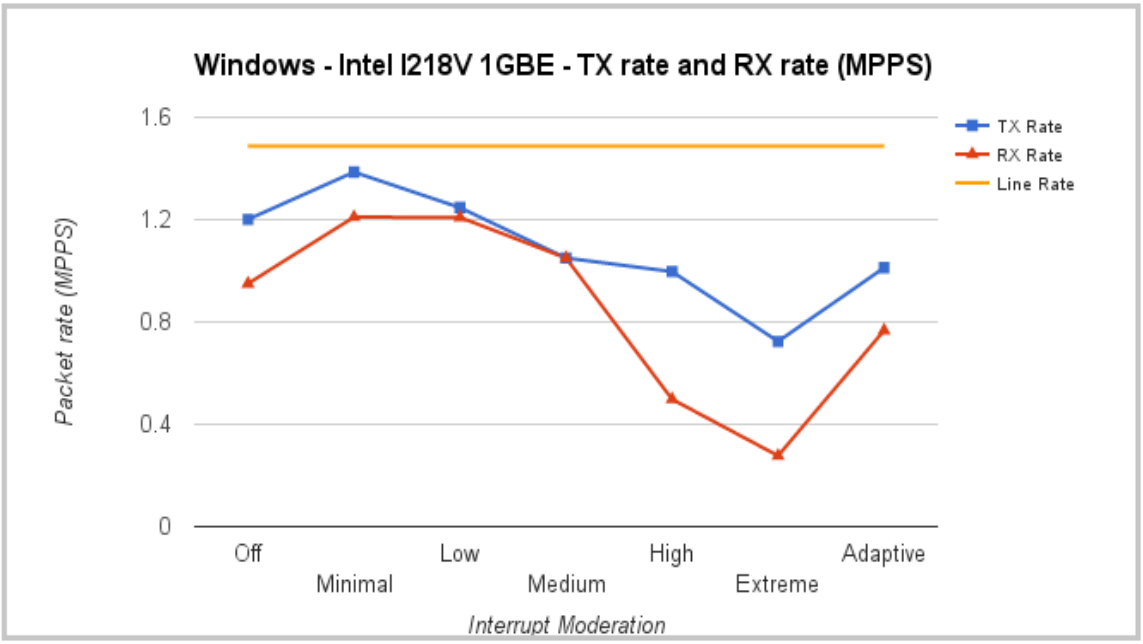


Figure 8.4: *TX/RX rates with different interrupt moderation values – Packet length 60 bytes*

With the interrupt moderation disabled the first problem that has been encountered is the crash of the system as soon as the RX rate passes the value of 0,95 MPPS: this likely happens because the number of generated interrupts goes over the number of interrupts manageable by the OS: this situation leads to an overflow of the kernel stack and a subsequent crash.

In the others values provided by the driver it is possible to see that augmenting the moderation rate of the interrupts cause a performance loss either in RX and in the TX side: this can be expected because an increase in moderation rate leads to an increased number of packets that will be stored in the NIC before an interrupt and so a subsequent read from the OS will be done.

8.1.3 *Comparison with different user-space applications*

To test the performances of the port of Netmap to Windows, a comparison with other publicly available applications has been done: the 2 chosen programs are Startrinity UDP test tool [18] and NetPerf [19].

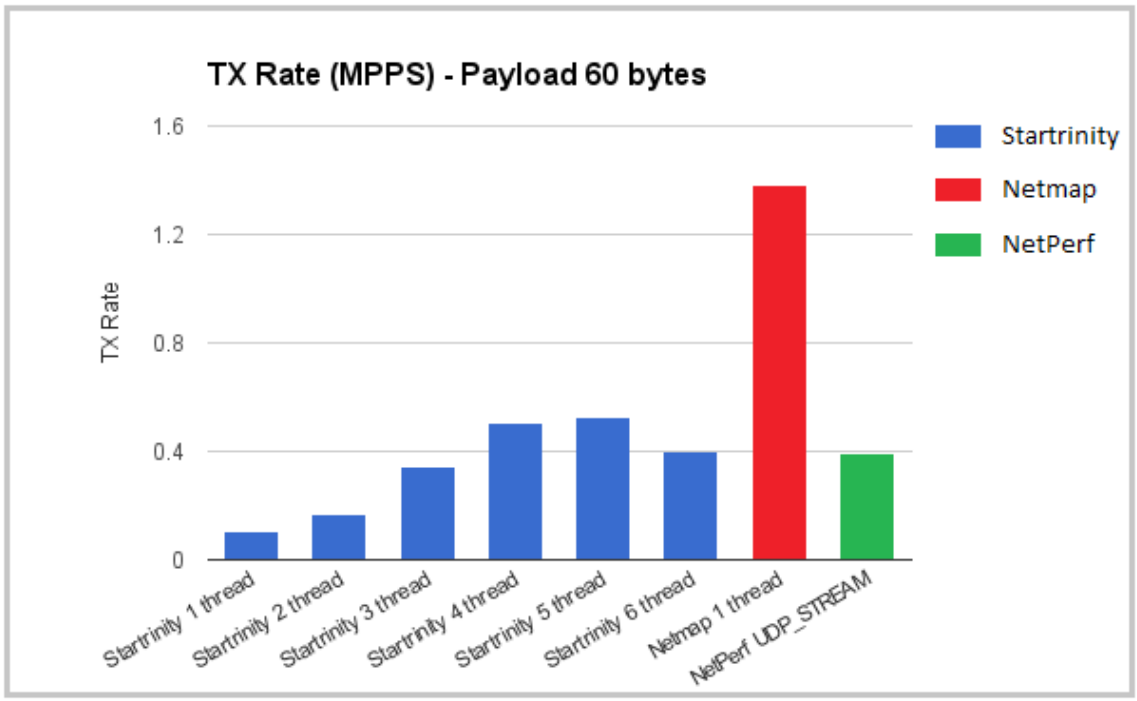


Figure 8.5: *Comparison of performances with other available applications*

Startrinity is an application used to test the reliability of a VoIP network and it is specialized in measurement of bandwidth, packet loss, jitter and delay; netperf is a tool to test the raw bandwidth of a network.

As is evident in Figure 8.5, Netmap has performances that cannot be compared with other applications, considering that with just one thread it is able to reach a rate three times better than the performance reached by NetPerf.

8.1.4 *Tests of tx/rx of data over a 10GbE link*

The first batch of tests has been made using Windows 8.1: this tests clearly show that even at low interrupt moderation rate, the OS cannot reach values higher than 1,85 MPPS.

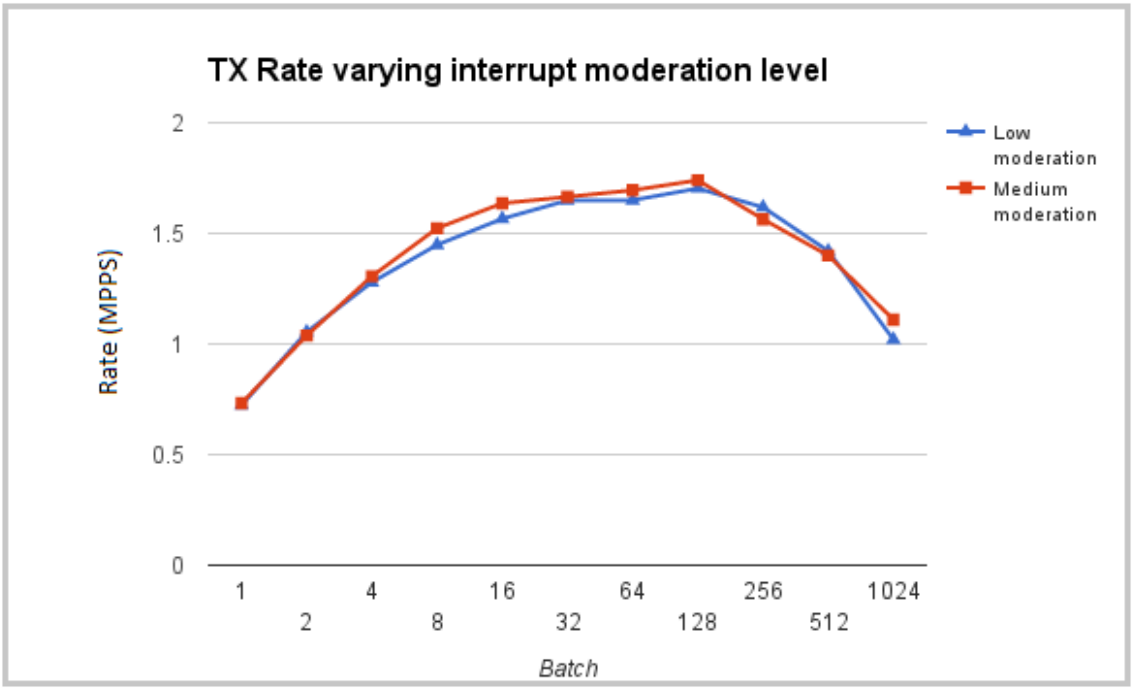


Figure 8.6: *TX Rate on Windows 8.1- 10 GbE – Packet size 60 bytes*

For this reason it has been chosen to test Netmap over a Windows Server 2012 R2 edition to test whether a difference could exists in the tuning of the kernel for interrupt handling and in terms of general performances between a home and a server edition of the Windows operating system.

The results emerged from the tests, clearly indicate that a Server edition of Windows can sustain an higher rate of transmission using the same hardware configuration; considering that the kernel of Windows 8.1 is the same of Windows Server 2012, an extensive optimization oriented to a high workload has probably been made to the kernel

to cope with the different needs of use respect to an home or individual professional use.

Figure 8.7: *Performances in Windows Server 2012 varying the interrupt moderation rate*

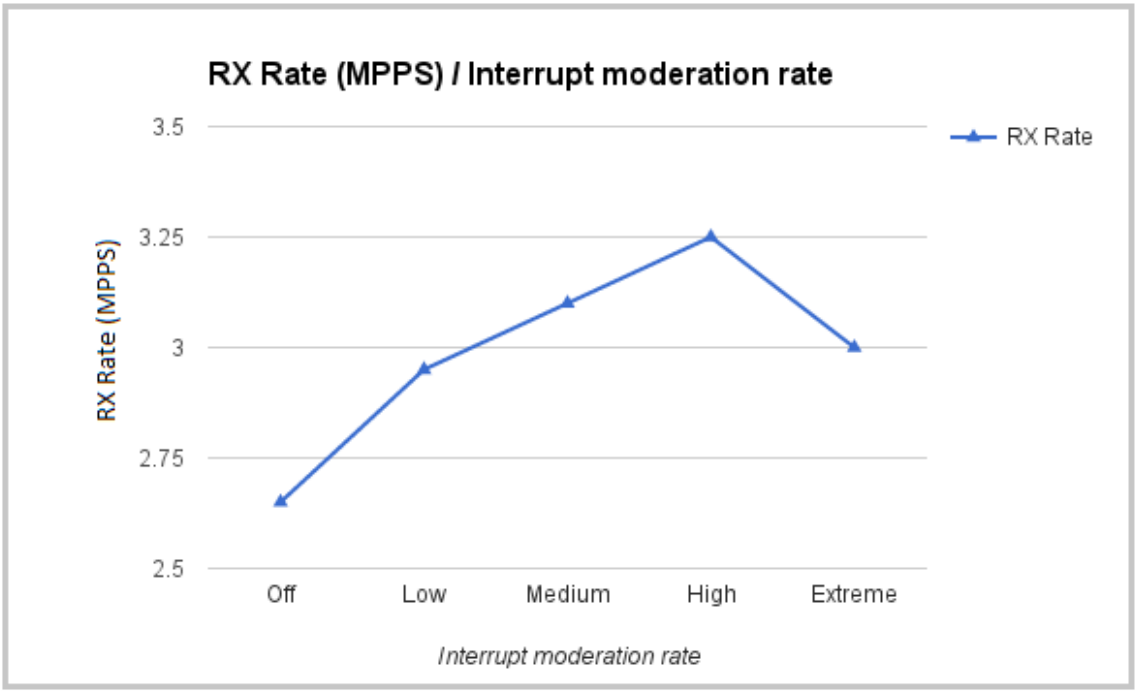


Figure 8.8: *RX performances in Windows Server 2012 respect to interrupt moderation*

The receive performances in figure 8.8 have been taken using the most performing batch (256 packets per send) and show the higher rate the OS can manage before becoming unstable; using rates over the limits shown in the figure leads to a loss of response of the terminal window running the pkt-gen application, then subsequently a loss of control of external peripherals like video card and input devices occurs; this as explained before, happens because the number of interrupt generated in the system overcomes the one actually manageable by the OS.

To conclude this series of experiment, it is useful to give some reference values to better understand the interrupt moderation mechanism: Intel provides those values in public available documentation [20], and those values are valid for every driver of every GbE NIC produced by Intel: the values are presented in the following table:

Interrupt moderation profile	Interrupt rate (int/sec)
None	1 interrupt every packet
Minimal	19000
Low	10000
Medium	4000
High	1900
Extreme	1000

8.1.5 *Overall comparison between different OS*

The tests have been made on the same machine with three different OS: FreeBSD, Windows 8.1 64bit and Windows Server 2012 R2 Datacenter edition 64 bit.

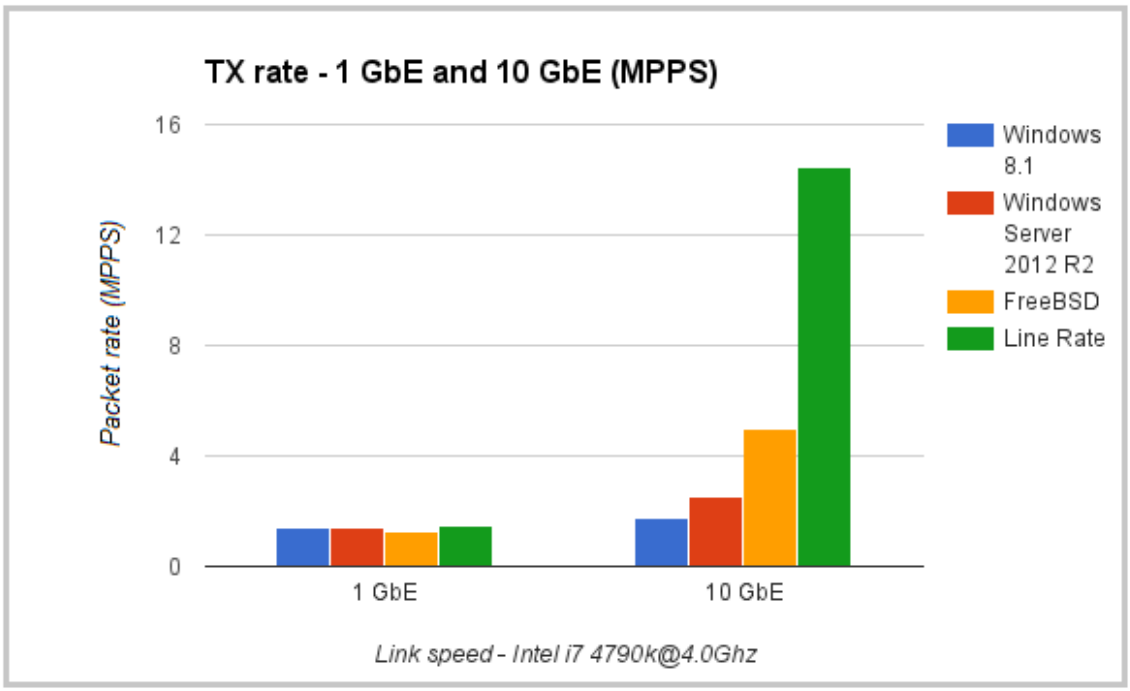


Figure 8.9: *Transmission results on different operating systems*

This series of tests show that the performances on different OSs are comparable using a 1GbE link, but on the 10GbE link the performances are almost a half respect to what is possible on FreeBSD with the same hardware and the same configuration of Netmap.

To better understand the nature of this limit, a last test has been made limiting the frequency of the CPU and of the cache by decreasing the value of the respective multipliers.

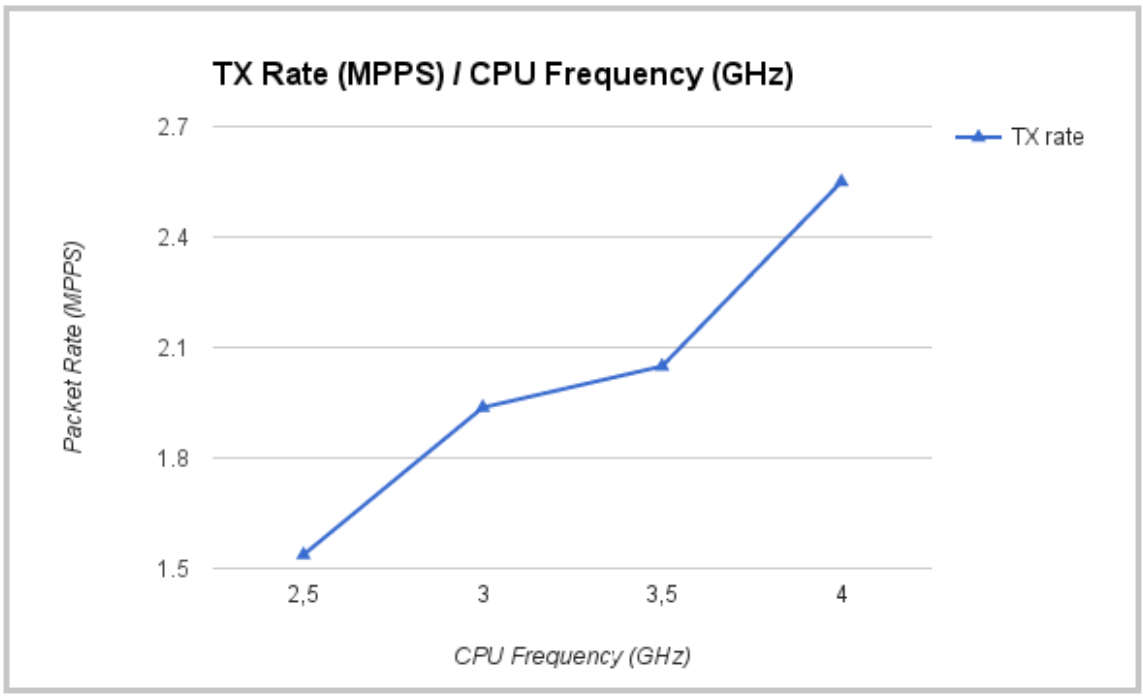


Figure 8.10: *Performances versus CPU frequency*

As figure 8.10 shows, this upper limit in transmission is capped by the frequency of the CPU, so to reach the same level of performances another set of optimization will be required to the core and to the NDIS modules.

8.2 Future works

Possible future works on the Windows port could involve developing a more powerful mechanism and an implementation of a zero-copy mechanism in the packet interception path.

For the poll() implementation, the Windows OS provides a mechanism called I/O Completion Port (IOCP) that could be used to implement a multiple file descriptor emulation of the POSIX environment Poll(). Another way, using Cygwin, could be to find a way to make the Cygwin dll call the already available IOCTL when a poll is issued from user-space programs.

For the zero-copy in the NDIS module, it can be implemented as a mechanism that indicate the owners of the intercepted packet to free the structures only when a packet has been already sent. Working in this way will decrease the number of copies per intercepted packet to only one and could increase once more the overall performance of the system.

Another step to be made to make Netmap perform even better under Windows OS could be to modify the original network devices drivers to be natively compatible with Netmap as it has been done in FreeBSD and Linux. The last step is harder respect to the other discussed because the source code of device drivers under Windows environment usually aren't freely available.

Bibliography

[1] Luigi Rizzo: Netmap - the fast packet I/O framework
http://info.iet.unipi.it/~luigi/netmap/

[2] Luigi Rizzo, Giuseppe Lettieri: VALE , a Switched Ethernet for Virtual Machines
http://info.iet.unipi.it/~luigi/papers/20121026-vale.pdf

[3] Microsoft Corporation: MSDN - Defining I/O Control Codes
https://msdn.microsoft.com/en-us/library/windows/hardware/ff543023.aspx

[4] Microsoft Corporation: MSDN – Using MDLs
https://msdn.microsoft.com/en-us/library/windows/hardware/ff565421.aspx

[5] Microsoft Corporation: MSDN – NDIS Drivers
https://msdn.microsoft.com/en-us/library/windows/hardware/ff565448.aspx

[6] Microsoft Corporation: MSDN – NET_BUFFER architecture
https://msdn.microsoft.com/en-us/library/windows/hardware/ff568377.aspx

[7] Microsoft Corporation: MSDN – Initializing a Filter Driver
https://msdn.microsoft.com/en-us/library/windows/hardware/ff554903.aspx

[8] Microsoft Corporation: MSDN – ExAllocatePoolWithTag Routine
https://msdn.microsoft.com/en-us/library/windows/hardware/ff544520.aspx

[9] Microsoft Corporation: MSDN - Using Lookaside Lists
https://msdn.microsoft.com/en-us/library/windows/hardware/ff565416.aspx

[10] Microsoft Corporation: MSDN - Managing Hardware Priorities
https://msdn.microsoft.com/en-us/library/windows/hardware/ff554368.aspx

[11] Microsoft Corporation: MSDN - Introduction to ERESOURCE Routines
https://msdn.microsoft.com/en-us/library/windows/hardware/ff548046.aspx

[12] Microsoft Corporation: MSDN - Service Functions
https://msdn.microsoft.com/en-us/library/windows/desktop/ms685942.aspx

[13] Microsoft Corporation: MSDN – INF file section and directives
https://msdn.microsoft.com/en-us/library/windows/hardware/ff547433.aspx

[14] Microsoft Corporation: MSDN – Using DIRDS
https://msdn.microsoft.com/en-us/library/windows/hardware/ff553598.aspx

[15] Microsoft Corporation: MSDN – INF DelService Directive
https://msdn.microsoft.com/en-us/library/windows/hardware/ff547377.aspx

[16] Microsoft Corporation: MSDN – INF AddService Directive
https://msdn.microsoft.com/en-us/library/windows/hardware/ff546326.aspx

[17] Microsoft Corporation: MSDN - Standardized INF Keywords for Network Devices
https://msdn.microsoft.com/en-us/library/windows/hardware/hh205409.aspx

[18] Startrinity UDP Test tool, Flood generator
http://startrinity.com/VoIP/NetworkTester/NetworkTester.aspx

[19] NetPerf, network testing tool
http://www.netperf.org/netperf/

[20] Interrupt Moderation, Intel® GbE Controllers: Application Note
http://www.intel.com/content/www/us/en/embedded/products/networking/gbe-controllers-interrupt-moderation-appl-note.html